CENG 3420
Computer Organization & Design

Lecture 12: Pipeline – Advanced

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

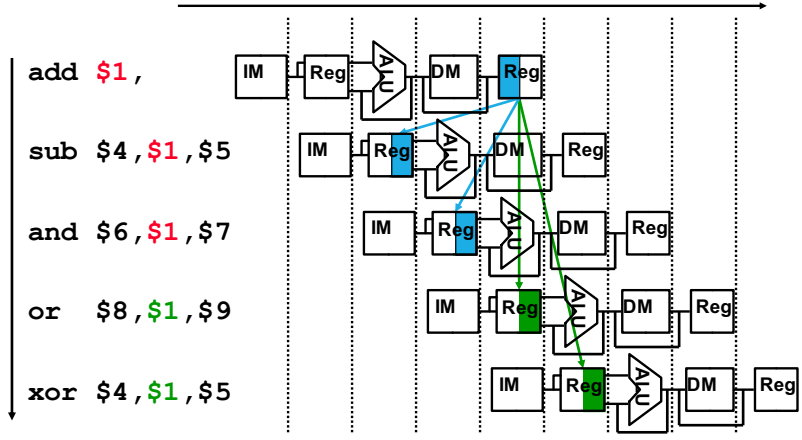
(Textbook: Chapters 4.7–4.9 & A.7–A.8)

Spring 2022

Data Hazards

Data Hazards: Register Usage

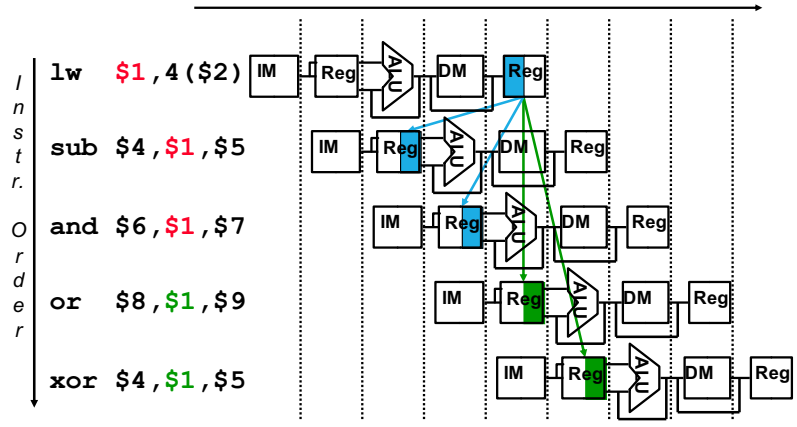
- Dependencies backward in time cause hazards



- Write After Read (WAR) data hazard

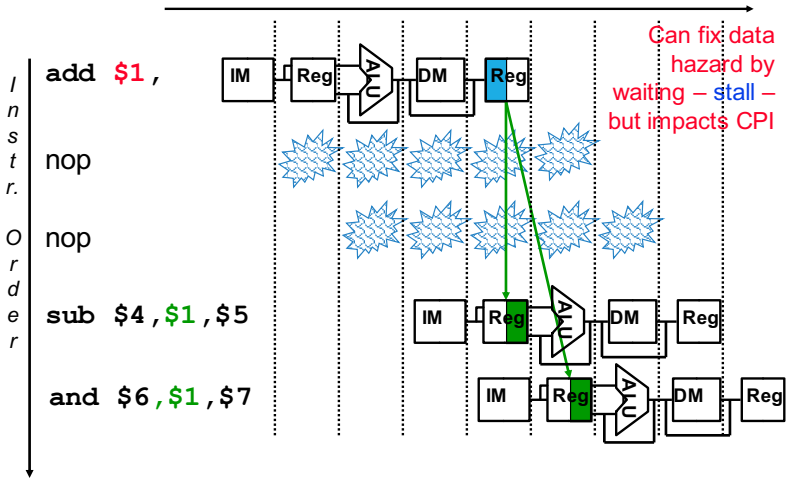
Data Hazards: Load Memory

- Dependencies backward in time cause hazards



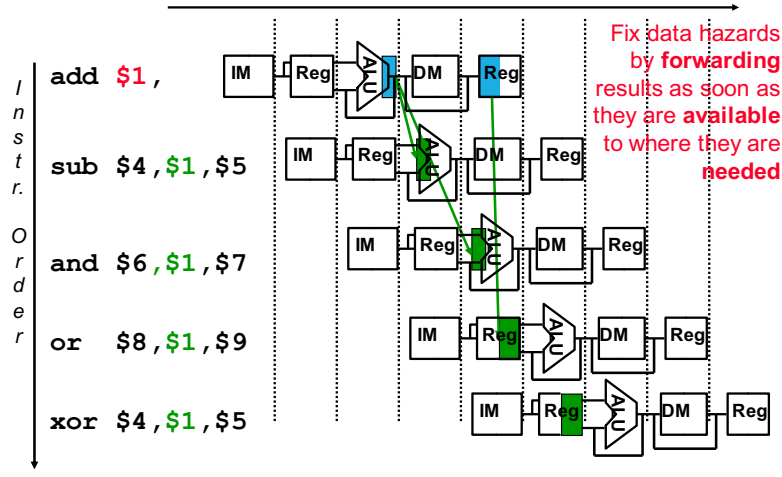
- Load-use data hazard

Resolve Data Hazards 1: Insert nop / stall



Resolve Data Hazards 2: Forwarding

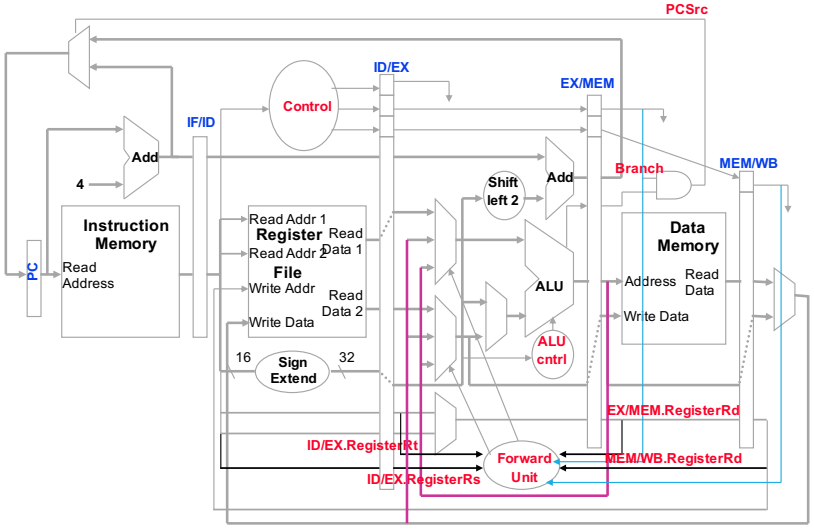
Fix data hazards by forwarding results as soon as they are available to where they are needed.



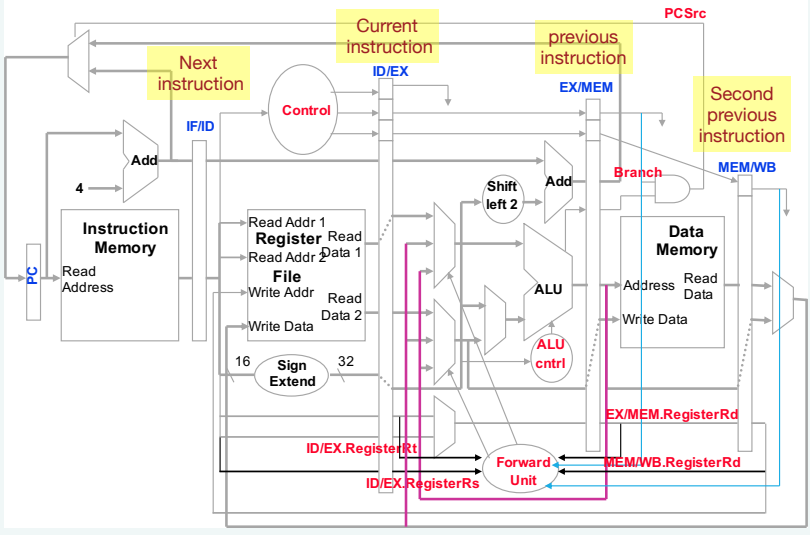
Forward Unit Output Signals

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Datapath with Forwarding Hardware



Note


PCSrc
Current instruction
previous instruction
Next instruction
ID/EX
EX/MEM
Second previous instruction
Control
IF/ID
Add
4
MEM/WB
Branch
Shift left 2
Add
Instruction Memory
Read Address
PC
Read Addr 1
Register File
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
ALU
ALU cntrl
Data Memory
Address
Read Data
Write Data
16
Sign Extend
32
EX/MEM.RegisterRd
ID/EX.RegisterRt
Forward Unit
MEM/WB.RegisterRd
ID/EX.RegisterRs

Data Forwarding Control Conditions

1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 10
```

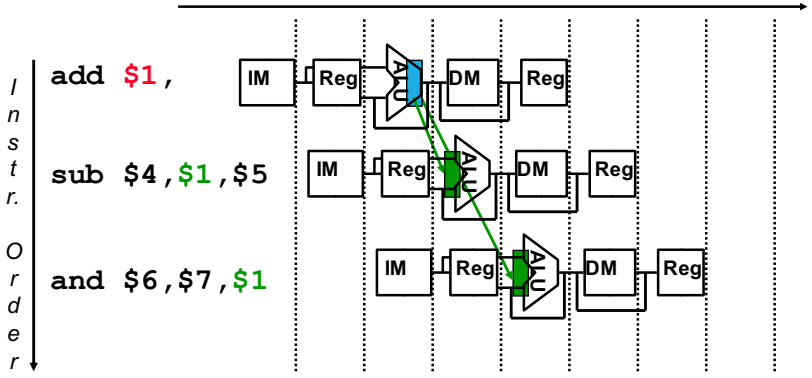
Forwards the result from the previous instr. to either input of the ALU

2. MEM Forward Unit:

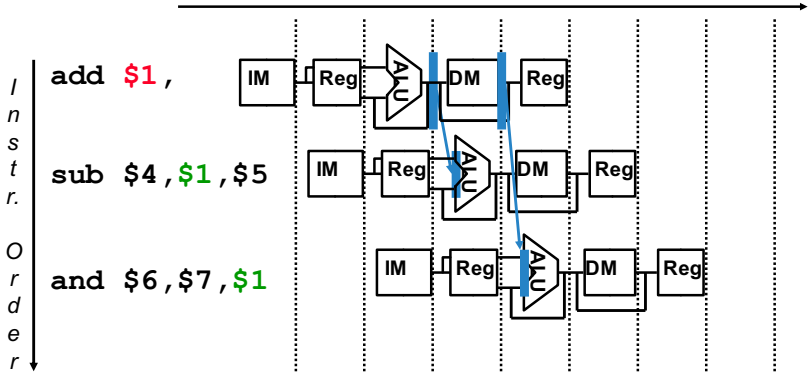
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the second previous instr. to either input of the ALU

Forwarding Illustration

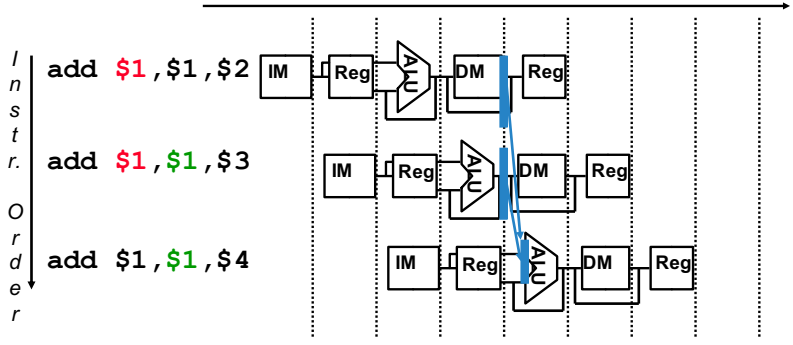


Forwarding Illustration



Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?



EX: Corrected MEM Forward Unit

- MEM Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)

and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)

and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 01
```

EX: Corrected MEM Forward Unit

- MEM Forward Unit:

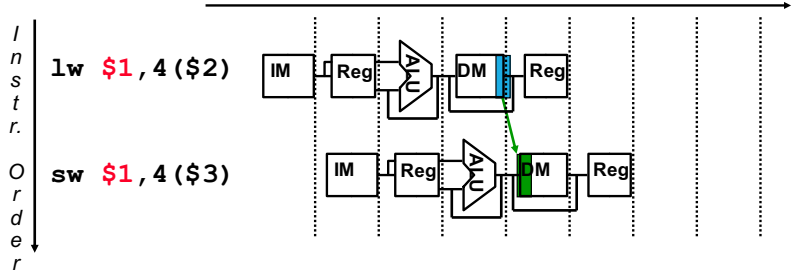
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 01
```

Note

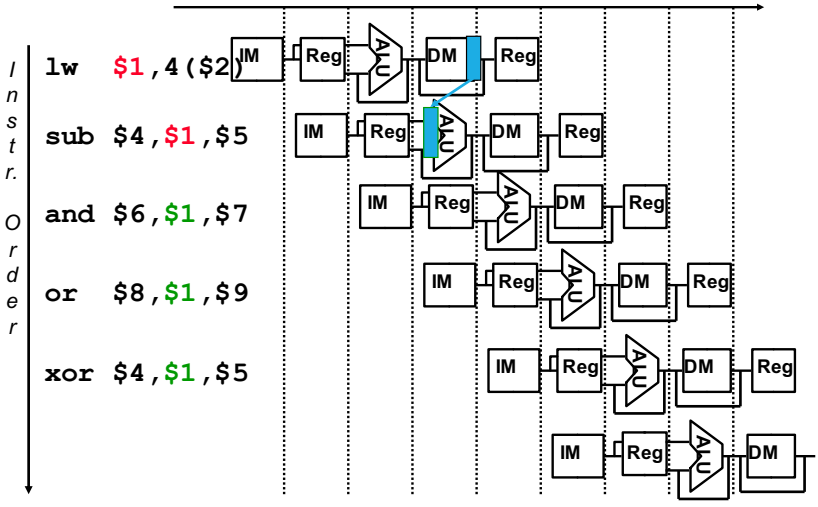
Compared to second previous instr (MEM/WB), the previous instr (EX/MEM) is with updated info!

Memory-to-Memory Copies

- For loads immediately followed by stores (memory-to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
- Would need to add a Forward Unit and a mux to the MEM stage

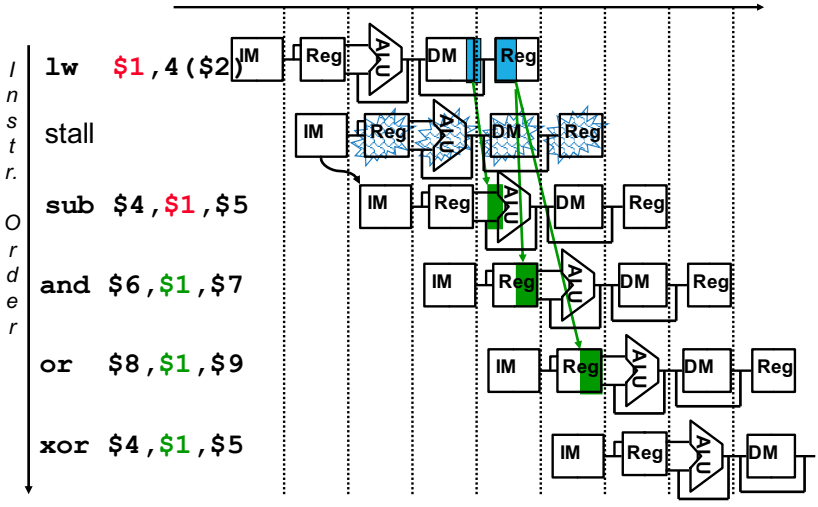


Forwarding with Load-use Data Hazards



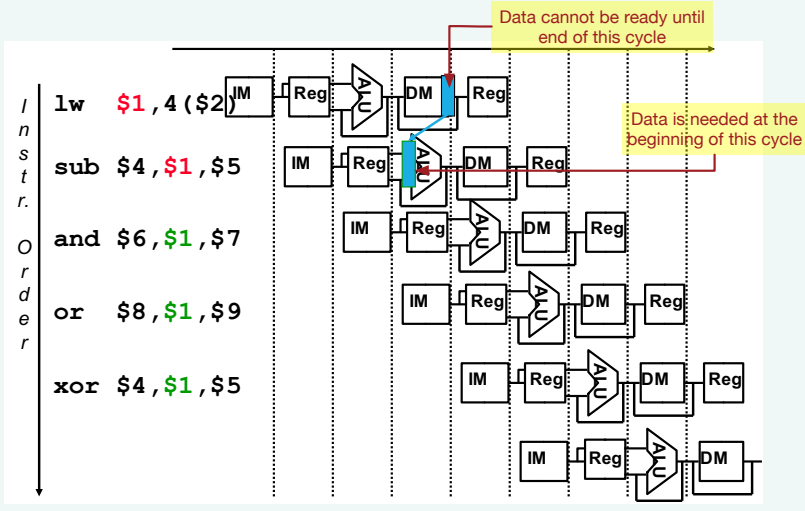
Will still need one stall cycle even with forwarding

Forwarding with Load-use Data Hazards



Will still need one stall cycle even with forwarding

Note: why have to stall?



Control Hazards

Control Hazards

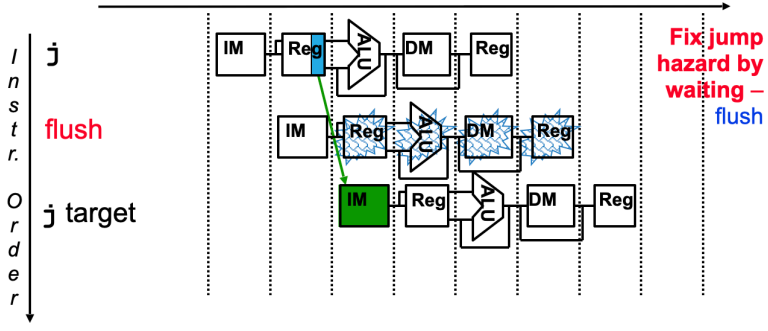
- When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Unconditional branches (`jal`, `jalr`)
  - Conditional branches (`beq`, `bne`)
  - Exceptions

- **Possible approaches:**
  - Stall (impacts CPI)
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - Predict and hope for the best !

Control hazards occur less frequently than data hazards, but there is nothing as effective against control hazards as forwarding is for data hazards

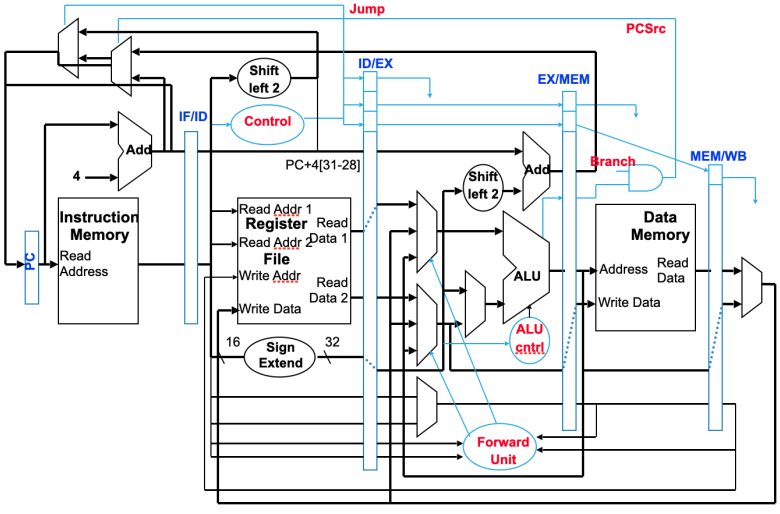
Control Hazards 1: Jumps Incur One Stall

- Jumps not decoded until ID, so one flush is needed
  - To flush, set IF.Flush to zero the instruction field of the IF/ID pipeline register (turning it into a nop)

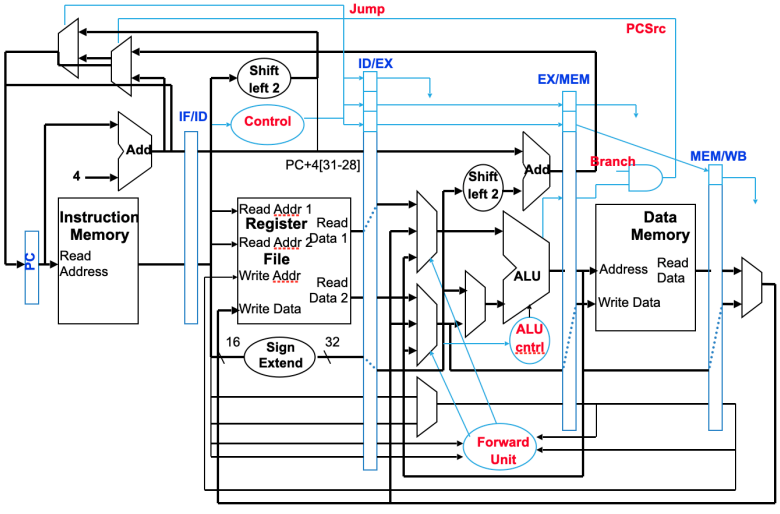


Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

Datapath Branch and Jump Hardware

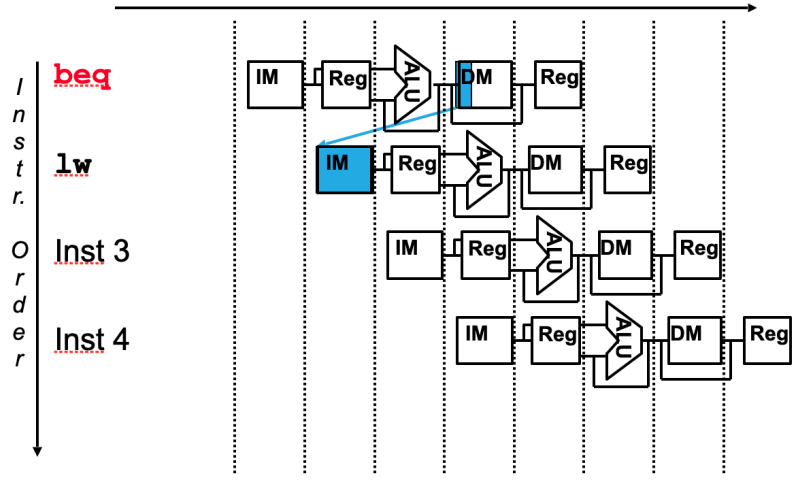


Supporting ID Stage Jumps

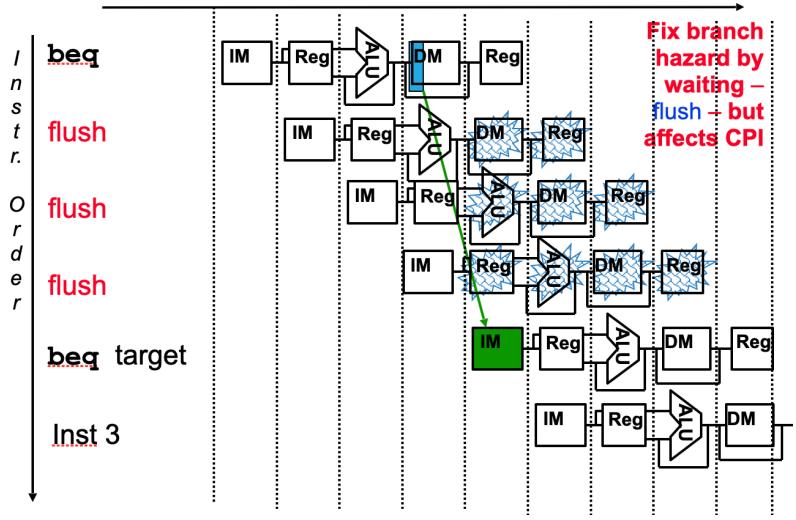


Control Hazards 2: Branch Instr

Dependencies backward in time cause hazards

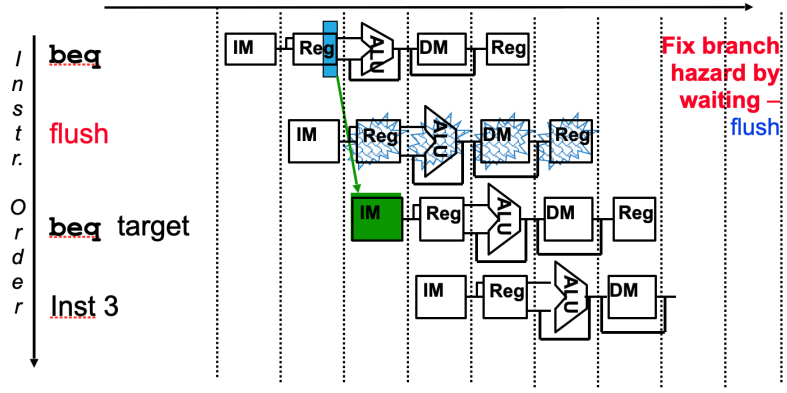


One Way to Fix a Branch Control Hazard



Another Way to Fix a Branch Control Hazard

Move branch decision hardware back to as early in the pipeline as possible – i.e., during the decode cycle



Note:

- Move branch decision hardware into ID stage
- What hardware? Arithmetic logics (comparator and adder)

Two Types of Stalls

- Nop instruction (or bubble) inserted between two instructions in the pipeline (as done for load-use situations)
  - Keep the instructions earlier in the pipeline (later in the code) from progressing down the pipeline for a cycle (bounce them in place with write control signals)
  - Insert nop by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- Flushes (or instruction squashing) were an instruction in the pipeline is replaced with a nop instruction (as done for instructions located sequentially after j instructions)
  - Zero the control bits for the instruction to be flushed

Note: Nop vs. Flush

Nop:

- determined in compilation stage
- inserted between instructions

Flush:

- determined during runtime
- an action to cancel one instruction that already in the pipeline.

Reducing the Delay of Branches

Move the branch decision hardware back to the EX stage:

- Reduces the number of stall (flush) cycles to two
- Adds an and gate and a 2x1 mux to the EX timing path

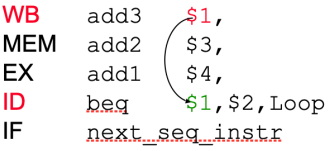
Add hardware to compute the branch target address and evaluate the branch decision to the ID stage:

- Reduces the number of stall (flush) cycles to one (like with jumps).
- But now need to add forwarding hardware in ID stage
- Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
- Comparing the registers can not be done until after RegFile read, so comparing and updating the PC adds a mux, a comparator, and an and gate to the ID timing path

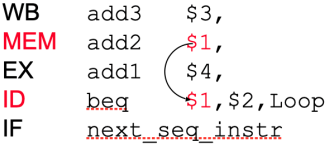
For deeper pipelines, branch decision points can be even later in the pipeline, incurring more stalls

ID Branch Forwarding Issues

- MEM/WB "forwarding" is taken care of by the normal RegFile write before read operation



- Need to forward from the EX/MEM pipeline stage to the ID comparison hardware for cases like




```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == IF/ID.RegisterRs))
       ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == IF/ID.RegisterRt))
       ForwardD = 1
```

Forwards the result from the second previous instr. to either input of the compare

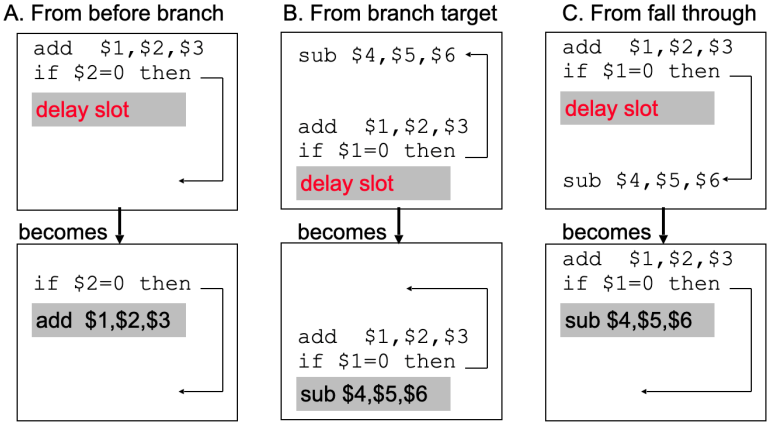
ID Branch Forwarding Issues

- If the instruction immediately before the branch produces one of the branch source operands, then a stall needs to be inserted (between the `beq` and `add`) since the EX stage ALU operation is occurring at the same time as the ID stage branch compare operation
  - Bounce the beq (in ID) and next_seq_instr (in IF) in place
  - Insert a stall between the add in the EX stage and the beq in the ID stage by zeroing the control bits going into the ID/EX pipeline register (done by the ID Hazard Unit)
- If the branch is found to be taken, then flush the instruction currently in IF (IF.Flush)

Delayed Branches

- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect after that next instruction
  - Compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay
- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot and reduces IC
- In B and C, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

Static Branch Prediction

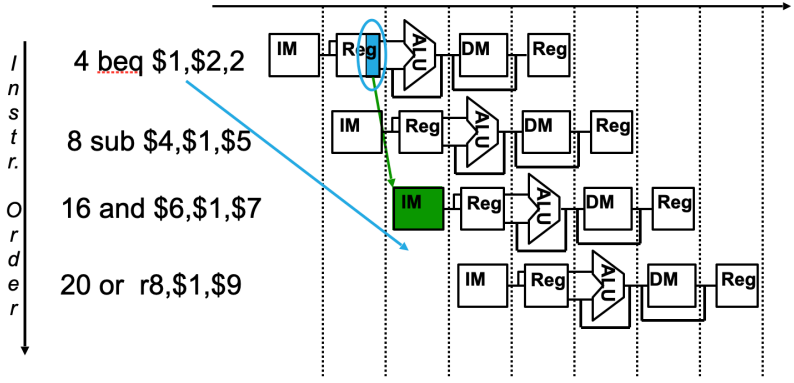
Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

Predict not taken

Always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch is taken does the pipeline stall

- If taken, flush instructions after the branch (earlier in the pipeline)
  - in IF, ID, and EX stages if branch logic in MEM – three stalls
  - In IF and ID stages if branch logic in EX – two stalls
  - in IF stage if branch logic in ID – one stall
- Ensure that those flushed instructions have not changed the machine state: automatic in the pipeline since machine state changing operations are at the tail end of the pipeline (e.g. `MemWrite` in MEM or `RegWrite` in WB)
- Restart the pipeline at the branch destination

Flushing with Misprediction (Not Taken)



To flush the IF stage instruction, assert IF.Flush to zero the instruction field of the IF/ID pipeline register (transforming it into a nop)

Branching Structures

- Predict not taken works well for "top of the loop" branching structures
- But such loops have jumps at the bottom of the loop to return to the top of the loop and incur the jump stall overhead
- Predict not taken doesn't work well for "bottom of the loop" branching structures

Top of the loop

```
while (condition) {
    func();
}
```

Bottom of the loop

```
do{
    func();
} while(condition);
```

Static Branch Prediction

Resolve branch hazards by assuming a given outcome and proceeding

Predict taken

predict branches will always be taken

- Predict taken always incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
- Is there a way to "cache" the address of the branch target instruction

As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution

Dynamic branch prediction

Predict branches at run-time using run-time information

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute
- Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but this doesn't affect correctness, just performance
  - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
- If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit(s)
  - A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

Branch Target Buffer

The BHT predicts when a branch is taken, but does not tell where its taken to!

- A branch target buffer (BTB) in the IF stage caches the branch target address, but we also need to fetch the next sequential instruction.
- The prediction bit in IF/ID selects which "next" instruction will be loaded into IF/ID at the next clock edge
- Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction

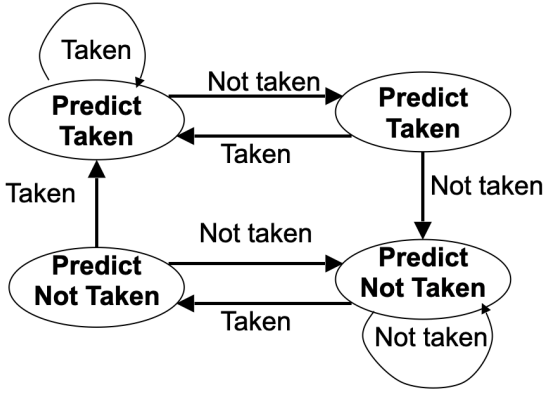
If the prediction is correct, stalls can be avoided no matter which direction they go

1-bit Prediction Accuracy

- A 1-bit predictor will be incorrect twice when not taken
  - Assume $predict_bit = 0$ to start (indicating branch not taken) and loop control is at the bottom of the loop code
  - First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit $(predict_bit = 1)$
  - As long as branch is taken (looping), prediction is correct
  - Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit $(predict_bit = 0)$
- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

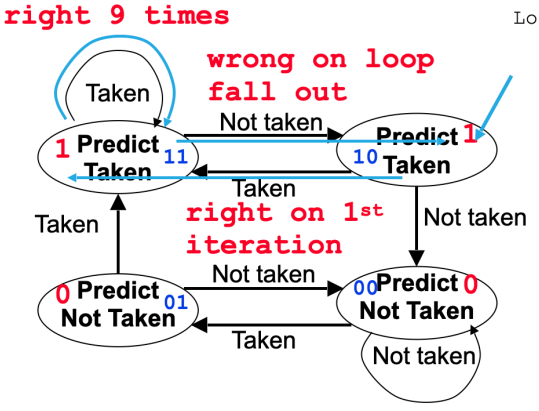
2-bit Predictors

A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed



2-bit Predictors

A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed




```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- BHT also stores the initial FSM state

Exceptions

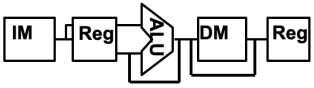
Dealing with Exceptions

- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- The software (OS) looks at the cause of the exception and deals with it

Two Types of Exceptions

- Interrupts – asynchronous to program execution
  - caused by external events
  - may be handled between instructions, so can let the instructions currently active in the pipeline complete before passing control to the OS interrupt handler
  - simply suspend and resume user program
- Traps (Exception) – synchronous to program execution
  - caused by internal events
  - condition must be remedied by the trap handler for that instruction, so much stop the offending instruction midstream in the pipeline and pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

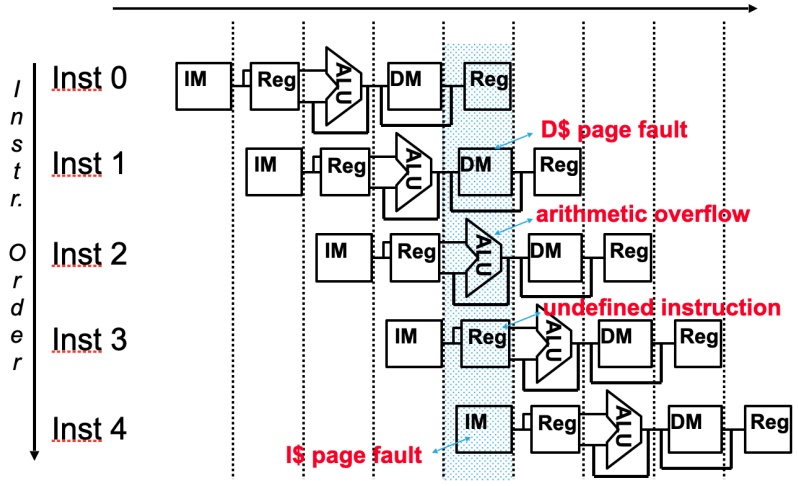
Where in the Pipeline Exceptions Occur



	Stage(s)?	Synchronous?
Arithmetic overflow	EX	yes
Undefined instruction	ID	yes
TLB or page fault	IF, MEM	yes
I/O service request	any	no
Hardware malfunction	any	no

- Beware that multiple exceptions can occur simultaneously in a *single* clock cycle

Multiple Simultaneous Exceptions



Hardware sorts the exceptions so that the earliest instruction is the one interrupted first

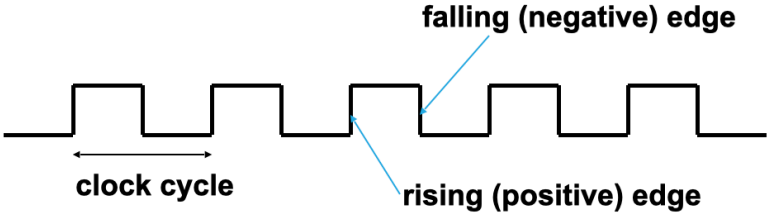
Summary

- All modern day processors use pipelining for performance (a CPI of 1 and a fast CC)
- Pipeline clock rate limited by slowest pipeline stage – so designing a balanced pipeline is important
- Must detect and resolve hazards
  - Structural hazards – resolved by designing the pipeline correctly
  - Data hazards
    - Stall (impacts CPI)
    - Forward (requires hardware support)
  - Control hazards – put the branch decision hardware in as early a stage in the pipeline as possible
    - Stall (impacts CPI)
    - Delay decision (requires compiler support)
    - Static and dynamic prediction (requires hardware support)
- Pipelining complicates exception handling

Background

Clocking Methodologies

- Clocking methodology defines when signals can be read and when they can be written



clock rate = 1/(clock cycle)
e.g., 10 nsec clock cycle = 100 MHz clock rate
1 nsec clock cycle = 1 GHz clock rate

- State element design choices
  - level sensitive latch
  - master-slave and edge-triggered flipflops

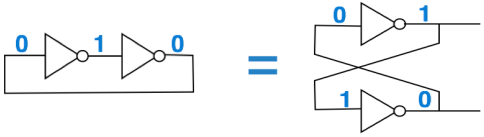
Review: Latches vs Flipflops

- Output is equal to the stored value inside the element
- Change of state (value) is based on the clock
  - Latches: output changes whenever the inputs change and the clock is asserted (level sensitive methodology)
    - Two-sided timing constraint
  - Flip-flop: output changes only on a clock edge (edge-triggered methodology)
    - One-sided timing constraint

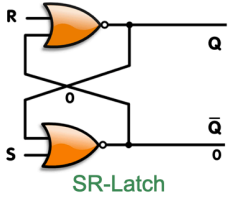
A clocking methodology defines when signals can be read and written – would NOT want to read a signal at the same time it was being written

Review: Design A Latch

- Store one bit of information: cross-coupled invertor



- How to change the value stored?



SR-Latch

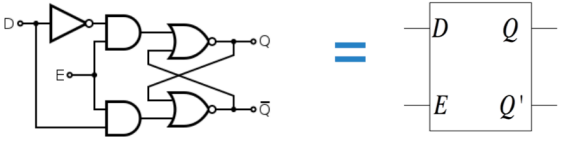
R: reset signal
S: set signal

S	R	Q	$\overline{Q}$
0	0	Qn	Qn
0	1	0	1
1	0	1	0
1	1	X	X

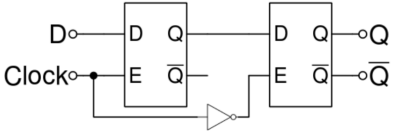
other Latch structures

Review: Design A Flip-Flop

- Based on Gated Latch

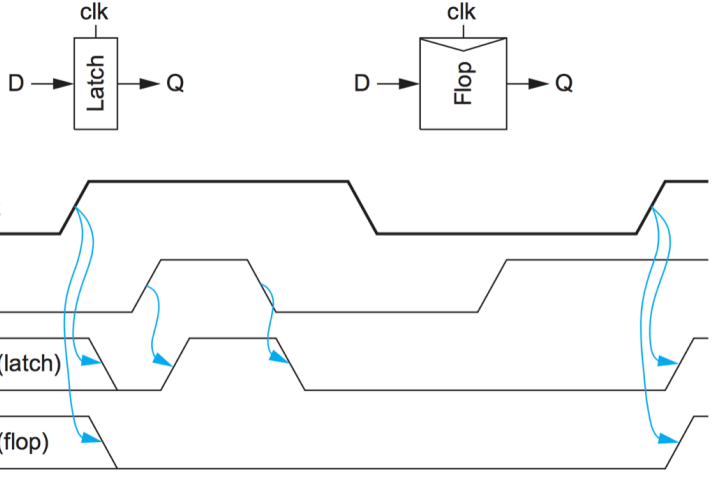


- Master-slave positive-edge-triggered D flip-flop



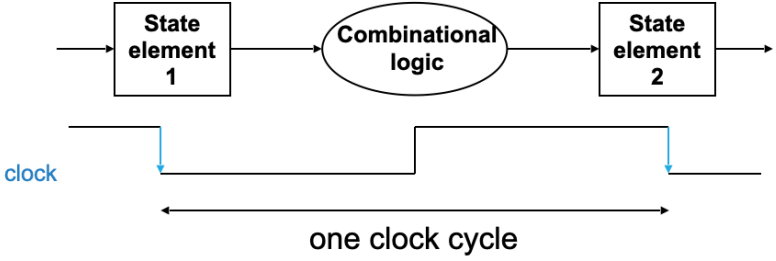
Review: Latch and Flip-Flop

- Latch is level-sensitive
- Flip-flop is edge triggered



Our Implementation

- An edge-triggered methodology
- Typical execution
  - read contents of some state elements
  - send values through some combinational logic
  - write results to one or more state elements



- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs